Improving BDD Based Symbolic Model Checking with Isomorphism Exploiting Transition Relations

Christian Appold
Chair of Computer Science V
University of Würzburg
Würzburg, Germany
appold@informatik.uni-wuerzburg.de

Symbolic model checking by using BDDs has greatly improved the applicability of model checking. Nevertheless, BDD based symbolic model checking can still be very memory and time consuming. One main reason is the complex transition relation of systems. Sometimes, it is even not possible to generate the transition relation, due to its exhaustive memory requirements. To diminish this problem, the use of partitioned transition relations has been proposed. However, there are still systems which can not be verified at all. Furthermore, if the granularity of the partitions is too fine, the time required for verification may increase. In this paper we target the symbolic verification of asynchronous concurrent systems. For such systems we present an approach which uses similarities in the transition relation to get further memory reductions and runtime improvements. By applying our approach, even the verification of systems with an previously intractable transition relation becomes feasible.

1 Introduction

The presence of concurrent software is steadily increasing due to the shift towards multi-core CPUs. This software consists of several parallel threads, which are executed asynchronously and interleaved. Some models for inter-thread communication exist, but the most flexible and prominent one is the use of fully shared variables. Well-known programming APIs like the POSIX pthread model or the WIN32 API support this model of communication. Unfortunately, concurrent software often is very error-prone, and bugs tend to be subtle and are hard to detect. Thus, to enable its use in safety-critical areas, reliable techniques to verify the correct operation of concurrent software are mandatory. One formal verification technique which has been proven to be successful in the verification of concurrent systems is temporal logic model checking [7], [18]. There, desired properties of a system are formulated in a temporal logic (like CTL [2] or LTL [16]), and the state-space of the system is investigated exhaustively to validate these properties. A very effective model checking technique is symbolic model checking [8], [12] based on Binary Decision Diagrams (BDDs) [3].

Nevertheless, BDD-based model checking is often still very memory and time consuming. This sometimes circumvents the successful verification of systems. The main reason for the large memory requirements of symbolic model checking is often the huge size of the BDD representing the transition relation. Therefore, some methods have been proposed to diminish this problem. Originally a monolithic transition relation consisting of a single BDD was used. Due to the large size of this BDD, the authors of [4] suggested to use partitioned transition relations. There, the transition relation is split into several pieces and each of these pieces can often be represented by a small BDD. Pieces of partitioned transition relations of asynchronous systems frequently possess many identity patterns for identity transformations of state variables. In [13] and [5] the removal of such identity patterns has been suggested to reduce the memory overhead. In this paper we target the symbolic verification of asynchronous concurrent

Giovanna D'Agostino, Salvatore La Torre (Eds.):
Proceedings of the Second International Symposium on
"Games, Automata, Logics and Formal Verification" (GandALF 2011)
EPTCS 54, 2011, pp. 17–30, doi:10.4204/EPTCS.54.2

© Christian Appold
This work is licensed under the
Creative Commons Attribution License.

systems, like e.g. concurrent software. We present a new memory saving approach to store the transition relation with BDDs. It allows to exploit similarities in the BDDs of the component transition relations. Additionally, identity patterns are removed, too. Furthermore, we introduce an algorithm that enables the efficient use of our new technique for model checking. Our experimental results show (see section 5) that this can lead to significant memory and runtime improvements. The approach is not restricted to asynchronous systems, but can be used for synchronous systems as well. To our knowledge, this is the first paper where similarities in the transition relation of components of a system are exploited that way.

The rest of this paper is organized as follows. In the next section we present some background information. We introduce our model of an asynchronous concurrent system (2.1) and give a short introduction into BDDs (2.2), symbolic state-space generation (2.3), and symbolic representations of transition relations and related work (2.4). Thereafter, Section 3 presents our new approach to store the transition relation and in Section 4 we exemplify an efficient algorithm to build the AND of an ordinary BDD and our new data structure. Experimental results which demonstrate the efficiency of our new approach can be found in Section 5. The paper closes with a conclusion and an outlook to future work.

2 Background

2.1 Asynchronous Concurrent Systems

In this paper we target finite state asynchronous concurrent systems $M^m = (S, R, S_0)$, where S is the finite set of possible states, $S_0 \subseteq S$ is the set of initial states and R is the transition relation. We assume that an asynchronous system M^m is composed of $m > 1$ components, and a state $s \in S$ is a tuple $s = (\vec{g}, l_1, ..., l_m)$. Thus, a system state consists of the values $\vec{g}$ of all global shared variables (not associated with any component) and the $local\, state\, l_i$ of each component $i \in \{1, ..., m\}$ (i.e. values of all local variables of component i). The transition relation is defined as $R = \{(x, x') |\; x \in S \wedge x' \in S \wedge$ state x' can be reached from state x in a single step$\}$.

The execution model of a system M^m is that of interleaved asynchrony. Only one component can execute a transition at a time and a transition of a component i only depends on and only changes the values of the shared variables $\vec{g}$ as well as its own local state l_i. That means, a component has neither read nor write access to local variables of other components. We denote this frequently occurring behavior as *transition locality*. Let R_{i_P} be a relation with $R_{i_P} = \{((\vec{g}, l_i), (\vec{g}', l_i'))|\; (\vec{g}', l_i')$ results from $(\vec{g}, l_i)$ by executing a single step of $i\}$ and let R_i be the transition relation of component i that contains the transitions executable by component i. In systems with transition locality the following holds $\forall i \in \{1, ..., m\} : R_i = \{((\vec{g}, l_1, ..., l_m), (\vec{g}', l_1', ..., l_m'))| \forall j \neq i :\; l_j' = l_j \wedge ((\vec{g}, l_i), (\vec{g}', l_i')) \in R_{i_P}\}$ and $R = \bigcup_{i \in m} R_i$. An example for this system type is the tremendous importance gaining concurrent software for multi-core architectures with threads which communicate via shared variables. Also the subtype of concurrent software with replicated threads is most relevant in practice. A formal definition of this system type can be found in [10].

2.2 Binary Decision Diagram (BDD)

Decision diagrams are used in symbolic model checking to store sets of states as well as the transition relation of a system. A *binary decision diagram* (BDD) [3] for N-variables can be used to encode a function $f : \{0, 1\}^N \mapsto \{0, 1\}$.

Definition 1. *A BDD is an acyclic directed graph with a single root vertex and two types of vertices, nonterminal vertices and terminal vertices. Each nonterminal vertex v is labeled by a variable* $var(v)$ *and has two successors* $low(v)$ *and* $high(v)$. *A terminal vertex v is labeled by a value* $value(v) \in \{0,1\}$.

As we did in this paper, most often *reduced ordered binary decision diagrams* ROBDDs [3] are used. ROBDDs are a canonical representation for boolean functions. Canonicity is achieved by using two restrictions for BDDs. There should be no isomorphic subtrees or redundant vertices in the diagram, and the variables should appear in the same order along each path from the root vertex to a *terminal* vertex. The same order for the variables along each path is ensured by using a total ordering $\prec$ on the variables that label the vertices in a BDD. Then $var(u) \prec var(v)$ is required for any vertex u in the diagram that has a *nonterminal* successor v. One can decide whether a particular truth assignment to its variables makes a function represented as a BDD true, or not, by traversing the graph from the root vertex to a terminal vertex. The value of a reached *terminal* vertex is the value of the function for the given variable assignment.

2.3 Symbolic State-Space Generation

As mentioned in the last section, BDDs are used in symbolic model checking to store sets of states as well as the transition relation of a system. A set of states Z can be encoded with a BDD through its characteristic function χ_Z. If the shared states $\vec{g}$ of an asynchronous system with m components can be encoded with n_g boolean variables and the local states of a component i with n_{l_i} boolean variables, then a BDD for $N = n_g + \sum_{i=1}^{m} n_{l_i}$ variables can be used to store sets of system states. To encode the transition relation with a BDD, transitions between states, instead of single states, have to be encoded. Therefore, a BDD for twice as many variables as for BDDs that encode sets of states is necessary and the transition relation can be encoded with a BDD for $2N$-variables. There N-variables are needed for the *from*-state and also N-variables for the *target*-state of a transition. As BDD variable ordering for the $2N$-variables, all possible permutations are applicable. But it is widely acknowledged that variable ordering with interleaving of the corresponding *from*- and *target*-state variables is often the most efficient variable ordering by terms of nodes required to store the transition relation. Thus, we consider only interleaved variable ordering in this work. In interleaved variable ordering the corresponding *from*- and *target*-state variables are next to each other in a BDD.

This paper targets on forward reachability analysis. There, the image computations are forward images and the forward image for a set of states Z is defined as: $Image(Z) = \{x' \mid \exists x \in Z, (x,x') \in R\}$. In forward reachability analysis state-space search starts with the set of initial states S_0. The set of reachable states is the minimal set satisfying $Z \supseteq S_0$ and $Z \supseteq Image(Z)$ which can be computed through iterated forward image calculations. The traditional approach for symbolic state-space generation, which we also used within this paper, uses breadth-first iterations. Each breadth-first iteration consists of an image computation with the entire transition relation R of a system. At the ith iteration all states with distance less or equal i from the initial states have been explored.

2.4 Symbolic Representations of Transition Relations and Related Work

A monolithic transition relation of a single BDD is often intractably large. Therefore, the use of partitioned transition relations has been proposed in [4]. Partitioned transition relations consist of conjunctions or disjunctions of a number of pieces of the single BDD. These pieces can often be represented by a small BDD. In this paper we consider asynchronous concurrent systems and use disjunctive partitioned

transition relations. A component-wise disjunctively partitioned transition relation for an asynchronous system with m components is composed of the transition relations R_i of the components, and can be written as $R = R_1 \vee R_2 \vee ... \vee R_m$. In this work we consider only systems with transition locality (see section 2.1). Our method further reduces the memory requirements of the partitioned transition relation approach through exploiting similarities in the transition relation of the components. For the use of partitioned transition relations, it's worth mentioning that a too fine granulated transition relation may not be the best choice. As long as the BDDs don't become too large, it is better to combine several transitions in one disjunct. In this way, fewer BDD nodes may be needed and also image calculation can possibly be accelerated. In [19] the authors presented and investigated an approach where the partitions of partitioned transition relations can consist of several transitions. Their experimental results confirm that larger partitions lead to big runtime savings. But they also observed an increase in the number of BDD nodes for coarser partitioned transition relations. By considering similarities in the transition relations of the components our approach allows to build much coarser partitions of transitions. Additionally, in the presence of large isomorphic subgraphs no strong increase in the total number of BDD nodes occurs. Thus, our approach can reduce the runtime without causing an increase of the memory requirements.

Transition relations of asynchronous systems often contain many identity patterns. As introduced in 2.1, if a component i executes a transition in a system with transition locality, then the local states for all other components $j \neq i$ remain unchanged. Therefore, the BDD for the transition relation R_i of component i contains identity patterns for the local state bits of all other components $j \neq i$. An example of an identity pattern can be found in Figure 1. There level k contains a vertex of a *from*-state and level $k+1$ a vertex of the corresponding *target*-state. According to Figure 1, if the vertices at level k and $k+1$ get assigned different values, then the BDD evaluates to 0. That means, if a BDD for a transition contains an identity pattern for a variable, the variable doesn't change its value when the transition is executed. To avoid the memory overhead to store identity patterns, [14] introduces an approach which uses reduced matrix diagrams (MxDs) [13] without identity nodes for the transition relation. The authors of [5] suggested to use a new identity reduction rule for MDDs [11] to get fully identity reduced MDDs for the transition relation. These papers just present approaches for identity reduction, but no method to use similarities in the transition relations of components. A technique to exploit sharing in BDDs for regular circuits that differ only in their support variables has been presented in [9]. Similar to our approach a remapping of input variables is used there. But such a remapping can not be used for BDDs of transition relations of components in asynchronous concurrent systems. The reason is different positions of identity patterns in the BDD variable ordering for different components. Additionally, they always expand a BDD with modified input variables before performing a BDD operation. This is very time consuming and can even be intractable for large transition relations. To solve this problem, we present in section 4 an efficient algorithm for boolean operation calculation with our new BDD type, which avoids the expansion to a normal BDD.

3 Transition Locality Exploiting BDDs (TLEBDDs)

In this section we present our new approach to store the transition relation of systems with transition locality (see section 2.1). It makes use of the circumstance that BDDs for subsets of the transition relation may have a very similar structure, if the transition relation is split component-wise in partitioned transition relations. To exploit those similarities and to reduce the memory requirements of transition relations we suggest to use *Transition Locality Exploiting BDDs (TLEBDDs)*. A TLEBDD consists of a normal BDD (see section 2.2) and a mapping list. For a system with m components, the transition

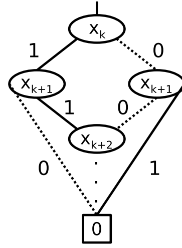

Figure 1: Example of an identity pattern

Bits Global Variables	Bits Local Variables			
	Component 1	Component 2		Component m
$b_{g1} \dots b_{g(2n_g)}$	$b_{11} \dots b_{1(2n_1)}$	$b_{21} \dots b_{2(2n_2)}$	...	$b_{m1} \dots b_{m(2n_m)}$

Figure 2: Variable Ordering Concatenated

relation of a component can be represented by a BDD with $2 \cdot N$ variables, where $N = n_g + \sum_{i=1}^{m} n_{l_i}$. In the rest of the paper we assume that the BDD of a TLEBDD for a component i is defined over the variables $X = \{x_1, x_2, ..., x_{2\cdot(n_g+n_{l_i})}\}$ and the mapping list is defined over the variables $Y = \{y_1, y_2, ..., y_{2\cdot N}\}$. We will denote the variables in X as reduced variables and the variables in Y as actual variables. The mapping list is necessary to map the reduced variables to the actual variables of the corresponding characteristic function χ_{R_i} of R_i for which the TLEBDD has been built. For a component i this mapping can be described with a function $\pi : \{1, 2, .., 2 \cdot (n_g + n_{l_i})\} \rightarrow \{1, 2, .., 2 \cdot N\}$ that maps mapping list entries to variable indices from Y.

Definition 2. *A n-mapping list is a list over Y with n elements, that is* $[y_{\pi(1)}, ..., y_{\pi(n)}]$.

According to section 2.1 the transition relation R_i of a component i in a system with transition locality is defined as $R_i = \{((\vec{g}, l_1, ..., l_m), (\vec{g}', l'_1, ..., l'_m)) | \forall j \neq i : l'_j = l_j \wedge ((\vec{g}, l_i), (\vec{g}', l'_i)) \in R_{i_P}\}$ (see section 2.1 for the definition of R_{i_P}) and the values of $\vec{g}'$ and l'_i depend only on $\vec{g}$ and l_i. TLEBDDs exploit the circumstance that for every transition of a component i holds $\forall j \neq i : l'_j = l_j$, and no vertices are used in the transition relation of a component i for the local states of an other component $j \neq i$.

Definition 3. *A TLEBDD for the transition relation of a component i in a system with transition locality is a tuple (G, b), where G is a normal BDD and b is a mapping list. G is a BDD with the $2 \cdot (n_g + n_{l_i})$ reduced variables $X = \{x_1, x_2, ..., x_{2\cdot(n_g+n_{l_i})}\}$. They are used for the bits of the shared states ($2 \cdot n_g$ bits) of the system and the local state bits ($2 \cdot n_{l_i}$ bits) of the component for which the TLEBDD has been built. For actual variables of the other $n-1$ components a TLEBDD implicitly assumes identity patterns. The mapping list b contains $n = \{1, 2, ..., 2 \cdot (n_g + n_{l_i})\}$ elements and is used to map the reduced variables of G to the actual variables. It contains for each position $q \in \{1, 2, ..., 2 \cdot (n_g + n_{l_i})\}$ in the variable ordering of the BDD G the associated actual variable $y_{\pi(q)}$. Thereby it holds for $q_1, q_2 \in \{1, 2, ..., 2 \cdot (n_g + n_{l_i})\}$ with $q_1 \neq q_2$ that $y_{\pi(q_1)} \neq y_{\pi(q_2)}$.*

TLEBDDs can be used for the efficient representation of component transition relations. A corresponding BDD can be obtained from a TLEBDD (G, b) through substitution of the reduced variables of the TLEBDD with the corresponding actual variables and the insertion of identity patterns. That means, the TLEBDD $(G, [y_{\pi(1)}, ..., y_{\pi(n)}])$ and the BDD $\Delta(G[y_{\pi(1)}/x_1, ..., y_{\pi(n)}/x_n])$ represent the same function.

Here $G[y/x]$ is the substitution of any occurrence of x in G with y and Δ is an operation which inserts identity patterns for associated pairs of from- and target-state actual variables for which no corresponding reduced variables exist. TLEBDDs use the same reduced variables to represent the local state bits of different components. In the prominent special case of asynchronous systems with only one replicated component type, even all corresponding local state bits of the m components can be mapped to the same reduced variables. In this way we get isomorphic subgraphs which aren't isomorphic in BDDs of ordinary partitioned transition relations, because the position of local state bits of components or of identity patterns in the variable ordering differs. This enables us to use the common property of BDD packages like Cudd [20] to store isomorphic subgraphs only once. Our experimental results in section 5 confirm that this can lead to enormous memory savings. TLEBDDs can be made canonical by requiring that mapping lists are ordered with respect to some strict ordering $\prec$ on the actual variables Y.

Definition 4. *A n-mapping list is ordered, if* $y_{\pi(i)} \prec y_{\pi(i+1)}$, *for all* $1 \leq i < n$.

Theorem 1. *If* (G, b_g) *and* (H, b_h) *are two TLEBDDs with mapping lists which are ordered with respect to some strict ordering* $\prec$ *on the actual variables* Y*, then for boolean functions* g, h *of component transition relations with* g *represented through* (G, b_g) *and* h *represented through* (H, b_h), $g = h$ *holds, if and only if* $G = H$ *and* $b_g = b_h$.

Proof. Let $b_g = b_h = [y_{\pi(1)}, ..., y_{\pi(n)}]$. By expansion of the TLEBDDs we get $g_{exp} = \Delta(G[y_{\pi(1)}/x_1, ..., y_{\pi(n)}/x_n])$ and $h_{exp} = \Delta(H[y_{\pi(1)}/x_1, ..., y_{\pi(n)}/x_n])$, where Δ is defined as introduced before. Because $G = H$, we get $g_{exp} = h_{exp}$ and therefore holds $g = h$.
Be now $g = h$. Because the mapping lists have to be strictly ordered and the same actual variables have to be mapped to reduced variables, there is only one unique ordered mapping list. Thus $b_g = b_h$ holds. If $G \neq H$ would hold, then the TLEBDDs (G, b_g), and (H, b_h) respectively, have to have different mapping lists b_g and b_h that $g = h$ can be valid. Therefore also $G = H$ holds. □

A TLEBDD can be built for a component i through encoding of the relation R_{i_P} by using the reduced variables instead of the actual variables. Additionally the mapping of the $2 \cdot (n_g + n_{l_i})$ reduced variables to the $2N = 2 \cdot (n_g + \sum_{i=1}^{m} n_{l_i})$ actual variables has to be stored in the mapping list. To evaluate the truth value of a particular assignment of values to the variables of a TLEBDD, its BDD has to be traversed from the root vertex to a terminal vertex similar to a BDD. Additionally, during its traversal the information which has been stored in the mapping list has to be considered to map the reduced variables to their corresponding actual variables and to take into account the missing identity patterns.

To use TLEBDDs and ordinary BDDs for model checking, it's necessary that they can be combined through boolean operations. An approach that allows the use of the traditional BDD algorithms to combine a TLEBDD and a BDD is to adapt the TLEBDD variable ordering to the variable ordering of the BDD and to insert simultaneously the omitted identity patterns. Though this works, here the uncompressed BDD has to be built for a TLEBDD. This would cause an additional runtime overhead, which can sometimes be very large. Also, if this BDD is huge a lot of memory may be required. In the worst case this can lead to an abort of the subsequent forward image calculation and therewith the model checking run. Therefore, we developed an effective algorithm for the calculation of boolean operations which avoids to generate normal BDDs for TLEBDDs entirely. In this way the vertices of a corresponding BDD for a TLEBDD are not needed at all, and we achieve the maximum possible memory reduction.

4 Efficient Algorithm for Boolean Operation Calculation

Here, we exemplify an efficient algorithm to compute the AND of a TLEBDD and a BDD. The AND of two BDDs is a very important step in forward image computation, because in every forward image computation the AND of the BDD with states which still have to be explored and the transition relation has to be calculated. Listing 1 sketches our new algorithm which allows to build the AND of a TLEBDD and a BDD without building the corresponding normal BDD for the TLEBDD at all. Prior to the execution of the algorithm the variable ordering of the reduced variables of the TLEBDD has to be adapted according to the variable ordering of the BDD .

Listing 1: Recursively compute the AND of a TLEBDD and a normal BDD

```
1  ANDRecursive(TLEBDDVertex TLEroot, BDDVertex BDDroot, int actualVarTLE){
2     BDDVertex result = TERMINAL_CASE(TLEroot,BDDroot,actualVarTLE);
3     if(result != NULL){
4        return  result;} //terminal case found
5     result = COMPUTED_TABLE_HAS_ENTRY(AND,TLEroot,BDDroot,actualVarTLE);
6     if(result != NULL){
7        return result;}  //result has already been calculated before
8
9     if(BDDroot.variable ≺ actualVarTLE){
10       v = BDDroot.variable;
11       T = ANDRecursive(TLEroot,BDDroot_v,actualVarTLE);
12       E = ANDRecursive(TLEroot,BDDroot_{\overline{v}},actualVarTLE);}
13    else{
14       v = actualVarTLE;
15       w = TLEroot.variable;
16       TLEroot_w = getNextVertex(TLEroot,TLEroot_w,actualVarTLE);
17       actualVarNew_w = getNextVertexVar(TLEroot,TLEroot_w,actualVarTLE);
18       TLEroot_{\overline{w}} = getNextVertex(TLEroot,TLEroot_{\overline{w}},actualVarTLE);
19       actualVarNew_{\overline{w}} = getNextVertexVar(TLEroot,TLEroot_{\overline{w}},actualVarTLE);
20       T=ANDRecursive(TLEroot_w,BDDroot_v,actualVarNew_w);
21       E=ANDRecursive(TLEroot_{\overline{w}},BDDroot_{\overline{v}},actualVarNew_{\overline{w}});}
22
23    if(T == E) return T;
24    R = FIND_OR_GENERATE_AND_ADD_UNIQUE_TABLE(v,T,E);
25    INSERT_COMPUTED_TABLE((AND,TLEroot,BDDroot,actualVarTLE),R);
26    return R;}
```

One main difference of the algorithm in Listing 1 to the usual AND algorithm is the use of a variable *actualVarTLE* for the current actual variable of a TLEBDD vertex. This variable is necessary to achieve that only those TLEBDD and BDD vertices are evaluated together that would also be evaluated together if the AND would be done between two ordinary BDDs. In line 2 of the algorithm it is detected if a terminal case of the recursive computation has been reached. If a terminal vertex is reached in a normal BDD, then its value is the value of the represented function for the variable assignment that led to this terminal vertex. In our algorithm a terminal vertex of a TLEBDD is really a terminal vertex, if its value is 0. If its value is 1, possibly missing identity patterns have to be evaluated before the terminal vertex is valid. This problem can be solved by using the value of *actualVarTLE* to decide the validity of such terminal vertices during the detection of terminal cases. The value of *actualVarTLE* also has to be considered during computed table accesses (see lines 5 and 25). This has to be done because different partial results of the AND operation can occur with the same TLEBDD and BDD vertices. By considering the value of

actualVarTLE these partial results can be differentiated.

Listing 2: Compute a successor vertex of the current TLEroot in a TLEBDD

```
1  getNextVertex(TLEBDDVertex TLEroot, TLEBDDVertex TLEroot_succ,int actualVarTLE){
2     TLEBDDVertex TLEroot_new = TLEroot_succ;
3
4     if(isTerminalVertex(TLEroot) ||
5       (actualVarTLE ≺ mappingList[TLEroot.variable])){
6         TLEroot_new = TLEroot;}
7
8     return TLEroot_new;}
```

Line 9 decides which of the two decision diagrams has the top variable in the used variable ordering at a step of the recursion. Adjustments to *actualVarTLE* and *TLEroot* for recursive calls of ANDRecursive have to be done only if *actualVarTLE* is the current top variable. Otherwise, its value is kept because the current root of the TLEBDD corresponds to an actual variable which has to be evaluated later. In the *else* path the new values of *actualVarTLE* ($actualVarNew_w$ and $actualVarNew_{\overline{w}}$) as well as *TLEroot* ($TLEroot_w$ and $TLEroot_{\overline{w}}$) have to be determined according to the current value of *actualVarTLE* and the mapping of the reduced variables of the TLEBDD vertices into the BDD variable ordering. Thereby the values of the new *TLEroot*s are calculated with the function *getNextVertex()* (see lines 16 and 18) and the new values of *actualVarTLE* are calculated with the function *getNextVertexVar()* (see lines 17 and 19). In the function *getNextVertex()* (see Listing 2) *TLEroot* has to keep its value, if it is already a terminal node, or if the value of *actualVarTLE* is before the actual variable that corresponds to the reduced variable of TLEroot in the variable ordering. This is necessary, because of the missing identity patterns in a TLEBDD, and *TLEroot* has to be evaluated later in the variable ordering. Otherwise *getNextVertex()* returns the successor $TLEroot_{succ}$ as the new root of the TLEBDD. The new value of *actualVarTLE* is calculated by the function *getNextVertexVar()* (see Listing 3). If the successor vertex $TLEroot_{succ}$ is a terminal vertex with value 0, then the terminal vertex can be evaluated immediately and *actualVarTLE* gets the value for a terminal vertex (see line 6). Otherwise, the function *identityPatternBeforeSuccVertex()* detects if there is an actual variable for an identity pattern between *actualVarTLE* and the corresponding actual variable of *TLEroot* in the variable ordering. If there is such an actual variable, the function *getNextActualIdentityPatternCurrVar()* calculates the next occurring actual variable of an identity pattern for a from-state and *actualVarNew* is set to this value.

These calculations can be done with the help of the mapping list and the parameter values of the functions *identityPatternBeforeTLEroot()*, and *getNextActualIdentityPatternCurrVar()* respectively. If no actual variable for an identity pattern exists in the variable ordering before the corresponding actual variable of *TLEroot*, *actualVarNew* can be set to a value for a terminal vertex if *TLEroot* is a terminal vertex. When *TLEroot* is no terminal vertex, *actualVarNew* is set to the value of an actual variable for an identity pattern before $TLEroot_{succ}$ or to the actual variable that corresponds to the formal variable of $TLEroot_{succ}$. By setting the value of *actualVarNew* to the first variable of every occurring identity pattern, we achieve that the recursion definitely holds at each such variable. The impact of the missing identity patterns then can be considered at these recursion steps.

Listing 3: Compute a new value for *actualVarTLE*

```
1  getNextVertexVar(TLEBDDVertex TLEroot , TLEBDDVertex TLEroot_succ ,int actualVarTLE){
2     int actualVarNew;
3
4     if((TLEroot_succ.index==CONST_INDEX) && (TLEroot_succ.value==0)){
5     {//a terminal vertex with value 0 can be evaluated immediately
6         actualVarNew = CONST_INDEX;}
7     else{
8        //decide if there is an identity pattern before TLEroot
9        //that has to be evaluated
10       if(identityPatternBeforeTLEroot(TLEroot,actualVarTLE)==TRUE){
11          actualVarNew =
12              getNextActualIdentityPatternCurrVar(TLEroot,actualVarTLE);}
13       else{
14          if(TLEroot.index==CONST_INDEX){
15            actualVarNew = CONST_INDEX;}
16          else{
17            if(identityPatternBeforeTLEroot(TLEroot_succ,actualVarTLE)==TRUE){
18              actualVarNew =
19               getNextActualIdentityPatternCurrVar(TLEroot_succ,actualVarTLE);}
20            else{
21               actualVarNew = mappingList[TLEroot_succ.variable];}}}}}
22
23    return actualVarNew;}
```

If a step of the recursion has finished, the calculated subgraphs T and E have to be combined and the result has to be returned. The return value is determined in lines 23 and 24 of Listing 1. If the top variable of the recursion step isn't a variable for an identity pattern, the return value can be calculated as it is done in the algorithm for the AND between two normal BDDs. When the top variable is a variable for an identity pattern, the recursion definitely holds at this recursion step and the variable is a from-state variable of the identity pattern. Here the impact of the missing identity patterns to the result of an AND operation is taken into account. Figure 3 illustrates the effect of identity patterns on the result calculation. In principle three different cases have to be considered. They are marked with a, b, and c, and x_k is the top and also from-state variable of an identity pattern. For each case the Figure shows in the left the result of the recursion at this step if the AND had been calculated with identity patterns. On the right side the result which our algorithm returns for TLEBDDs is shown. Except for the first case (a), two subgraphs are shown as solutions for our algorithm. There are two different subgraphs because of different optimizations that we used. Generally, after forward image calculations first the from-state variables are existentially abstracted and after that the target-state variables are shifted to their corresponding from-state variables. This is done with two different functions calls. Beneath the image calculation itself, these functions often need a lot of runtime. If TLEBDDs are used the abstraction of the from-state variables can be done easily and with little runtime overhead for variables for identity patterns. To do this there have to be inserted no vertices for the from-state level but only the correct remaining subgraph without the from-state vertex has to be built. Therefore we developed a version where from-state variables for identity patterns are abstracted away immediately. The outcome of the result combination with this immediate abstraction are captioned with *exist abst.* in Figure 3. Also we observed that the shift to the from-state variables often needs a lot of runtime. We developed a second method for result combination, where the target-state variables are immediately shifted to their corresponding from-state variables. This can be done easily for identity patterns. For the verification experiments we implemented the immediate shift for all variables. For non identity variables there is

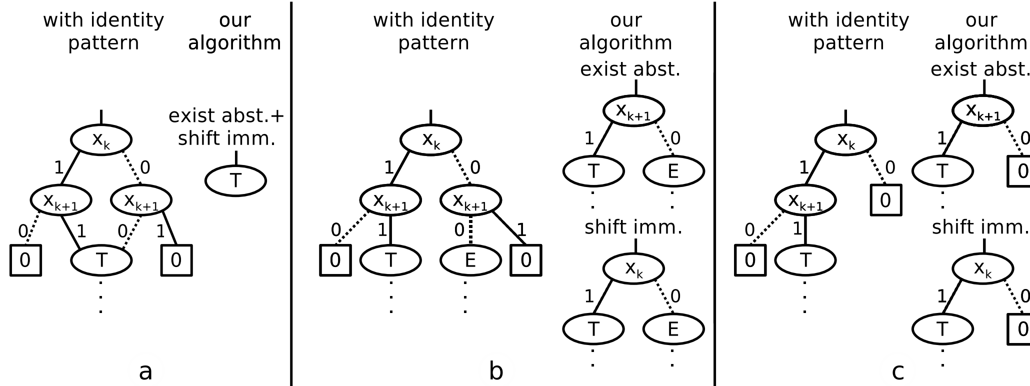

Figure 3: Handling of identity patterns during combination of the subgraphs T and E

more work to do to get the correct subgraphs. As our experimental results show, the immediate shift leads to very large runtime and memory improvements. Thus interleaved variable ordering is very efficient in combination with the immediate shift to the from-state variables. In the first case (a) the target-state vertices at level $k+1$ have as one successor the same subgraph T. This corresponds to the case where T and E are equal in our algorithm (see line 23 in Listing 1). When T equals E, the subgraph T can be returned regardless if an immediate abstraction of the from-state variables or an immediate shift to the from-state variables was done. If T and E are not equal, the result is calculated in line 24. Here two different cases can occur in the presence of identity patterns. The one is numbered with (b) in Figure 3 and there different subgraphs T and E exist for the identity paths. After abstraction of the from-state variables the subgraph with root variable x_{k+1} is the correct result. If an immediate shift is done, the result is the subgraph with root x_k. In the last case (c) only a system state exists for the value 1 of x_k for the current variable assignment (the same behavior can occur with value 0 for x_k). Here our algorithm also returns the subgraph with root x_{x+1} or x_x in dependency of the chosen result combination strategy. After the result has been calculated for a recursion step, it is inserted into the computed table (see line 25) and returned.

5 Experimental Results

In this section we present the results of our verification experiments. The experiments run on an Intel Pentium Core 2 CPU with 2.4 GHz and 3 GB main memory by using a single core. The verification experiments have been done with an adapted version of the symbolic model checker Sviss [21], which uses the Cudd BDD package [20]. For our experiments we have chosen the variable ordering concatenated for the bits of the components in the BDDs, because it is efficient for asynchronous systems. Figure 2 presents this variable ordering. The first bits in this variable ordering are b_{g1} to $b_{g(2n_g)}$. They denote the from-state and target-state bits for the shared variables of a system state. The bits b_{ij} denote the jth bit of component i. All experiments have been done with partitioned transition relations with identical sets of transitions in every partition for the different transition relation types. All testcases describe asynchronous systems with replicated components. In the following tables the number of replicated components can be found in the column *Problem* after the name of the verification benchmark. *Number of BDD Nodes* is the largest number of live BDD nodes that appeared during a verification experiment.

This is the memory bottleneck of a verification experiment, because the model checker has to store this number of BDD nodes to finish verification successfully [22]. *Time* is the runtime of a verification experiment, where s, m and h are abbreviations for seconds, minutes and hours. In Table 1 we show experimental results for forward reachability analysis. Experimental results for a standard partitioned transition relation, a TLEBDD as transition relation (from-state variables are abstracted immediately for identity patterns here (see section 4)) and a TLEBDD as transition relation where we immediately shift the target-state variables to its corresponding from-state variables are presented there. With the immediate shift we achieve significant runtime improvements and the memory gain can be maximized. One reason for the memory gain is that vertices which can be saved in a TLEBDD are not needed for the intermediate result BDD before the shift to the from-state variables. For the experiments in Table 1 we used a timeout of 24 hours.

Table 1: Verification results for forward reachability analysis

	Ordinary Partitioned Transition Relation		Transition Relation with TLEBDDs		Transition Relation with TLEBDDs and shift to target-state immediately	
Problem	Number of BDD Nodes	Time	Number of BDD Nodes	Time	Number of BDD Nodes	Time
MutexLocal 5	252,176	34s	176,577	29s	140,808	3s
MutexLocal 7	6,618,487	47:59m	4,977,342	44:05m	4,090,041	5:37m
MutexLocal 8	41,448,929	7:45h	31,092,345	7:10h	25,704,013	51:37m
Peterson 5	1,470,096	6:32m	720,661	5:28m	577,274	49s
Peterson 6	11,051,785	3:20h	8,562,251	3:20h	6,344,196	24:40m
Peterson 7	>100,000,000	>24h	>100,000,000	>24h	89,401,785	10:46h
CCP 5	205,449	1:02m	172,964	57s	117,875	4s
CCP 8	9,840,064	4:49h	9,118,465	4:35h	5,855,155	16:23m
CCP 10	>75,000,000	>24h	>75,000,000	>24h	67,822,819	12:32h
DP 15	309,329	3:55m	294,403	3:50m	193,402	16s
DP 20	3,614,204	2:27h	3,539,148	2:27h	2,267,828	8:41m
DP 22	9,595,403	9:06h	9,446,319	9:11h	6,018,632	32:34m

The first benchmark in Table 1 is an extended simple Mutual Exclusion Algorithm. There, a critical section exists which can be reached by a component if a shared variable points to it. This benchmark has also other shared variables. They store for every control state of the components the number of components currently being in this control state. Additionally, every component has one local variable which stores the number of components currently being in the new control state when a component moves its control state. Our experimental results show that big memory improvements can be achieved by using our TLEBDD to store the transition relations and we also see slight runtime improvements. The runtime improvements occur with TLEBDDs because we don't have to walk through edges of identity patterns in the recursion by using our new algorithm ANDRecursive. If we additionally shift the state variables immediately to the corresponding from-state variables, we even get further memory reductions and also large runtime improvements. The second testcase in Table 1 is the Peterson Mutual Exclusion Protocol [15]. It is a protocol where entry to the critical section is gained by a single process via a series of $n-1$ competitions. There is at least one loser for each competition and the protocol satisfies the mutual

exclusion condition, since at most one process can win the final competition. Table 1 shows that we achieve significant memory gains by just using TLEBDDs as transition relation. If we additionally shift the state variables immediately, we can further reduce the peak number of live nodes and we get very large runtime improvements. Table 1 also shows experimental results for the CCP Cache Coherence Protocol. It refers to a cache coherence protocol developed from S. German (see for example [17]). As our experimental results show, we can slightly reduce the memory requirements by using a TLEBDD. When we shift the state variables immediately, we get significant additional memory and runtime improvements. The last testcase in Table 1 is the Dining Philosophers Problem (mentioned DP in Table 1). Our implementation is an imitation of the monitors solution from [1]. The experimental results show that the memory requirements can not be reduced very much by using TLEBDDs. Also a little runtime increase can be observed for 22 components. This runtime increase can presumably be eliminated by optimizing the cache utilization. Nevertheless, significant memory and runtime savings can be observed again when we shift the state variables immediately.

Table 2: Experimental results for building only the transition relation

	Ordinary Partitioned Transition Relation	Transition Relation only identity reduced	Transition Relation with TLEBDDs
Problem	Number of BDD Nodes	Number of BDD Nodes	Number of BDD Nodes
MutexLocal 75	115,735,537	10,105,558	141,623
MutexLocal 255	mem ov	77,576,543	320,755
MutexLocal 2047	mem ov	mem ov	3,414,118
Peterson 8	110,560,066	47,415,495	17,403,225
Peterson 9	mem ov	115,675,330	40,089,105
Peterson 10	mem ov	mem ov	90,597,275
CCP 18	74,758,155	74,728,268	9,840,393
CCP 19	mem ov	mem ov	19,671,729
CCP 21	mem ov	mem ov	78,656,120

Table 2 shows experimental results about the maximum number of components for which the transition relation can be built alone with different transition relation types. We there present experimental results for a standard partitioned transition relation, a partitioned transition relation which is only identity reduced and for a transition relation with TLEBDDs. As our experimental results show, the number of components for which the transition relation can be built can always be enlarged by using TLEBDDs. If we use only identity reduction, we can not increase the number of components as large as with TLEBDDs and we even don't get an increase in the number of components for the CCP testcase. This shows the efficiency of our TLEBDD approach. We omitted the experimental results for the dining philosophers testcase here, because it only has a small transition relation that can already be build with an ordinary partitioned transition relation for more than 1000 components.

6 Conclusion and Outlook

In this paper we presented a new approach to store the transition relation of asynchronous systems. Our approach exploits the common property of BDD packages to store isomorphic subgraphs only once. The

presented experimental results confirm that our approach can lead to big memory savings. This allows the verification of larger systems. Additionally, our method can enlarge the parts of the transition relation which can be stored in a single partition of a partitioned transition relation. In this way fewer nodes may be needed and verification can possibly be accelerated. Additionally, we presented a new algorithm to combine BDDs and TLEBDDs efficiently. As our experimental results confirm, an immediate shift to the from-state variables leads to very large runtime and memory reductions for interleaved variable orderings by using this new algorithm.

In the future we intend to investigate the usage of TLEBDDs for storing the transition relation with other state-space exploration algorithms than the traditional breadth-first algorithm. By using other algorithms, like e.g. breadth-first generation with chaining, or the saturation algorithm, possibly even greater memory savings may occur. To investigate the performance of the use of TLEBDDs with other verification benchmarks and state-space exploration algorithms we intend to implement their usage for the symbolic model checker NuSMV [6]. Also, we will try to investigate the consequences of different TLEBDD variable orderings on the memory requirements and the verification runtime.

References

[1] M. Ben-Ari (2006): *Principles of Concurrent and Distributed Programming (2nd Edition) (Prentice-Hall International Series in Computer Science)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[2] M. Ben-Ari, Z. Manna & A. Pnueli (1981): *The temporal logic of branching time*. In: *POPL '81: Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ACM, pp. 164–176, doi:10.1145/567532.567551.

[3] R. E. Bryant (1986): *Graph-Based Algorithms for Boolean Function Manipulation*. *IEEE Transactions on Computers* 35, pp. 677–691, doi:10.1109/TC.1986.1676819.

[4] J. R. Burch, E. M. Clarke & D. E. Long (1991): *Symbolic Model Checking with Partitioned Transition Relations*. North-Holland, pp. 49–58.

[5] G. Ciardo & A. J. Yu (2005): *Saturation-based symbolic reachability analysis using conjunctive and disjunctive partitioning*. In: *Proc. CHARME, LNCS 3725*, Springer-Verlag, pp. 146–161.

[6] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani & A. Tacchella (2002): *NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking*. In: *Proc. International Conference on Computer-Aided Verification (CAV 2002), LNCS* 2404, Springer, Copenhagen, Denmark.

[7] E. M. Clarke & E. A. Emerson (1982): *Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic*. In: *Logic of Programs, Workshop*, Springer-Verlag, London, UK, pp. 52–71, doi:10.1007/BFb0025774.

[8] E. M. Clarke, O. Grumberg & D. A. Peled (2000): *Model checking*. MIT Press.

[9] A. Goel, G. Hasteer & R. Bryant (2003): *Symbolic representation with ordered function templates*. In: *Proceedings of the 40th annual Design Automation Conference*, DAC '03, ACM, New York, NY, USA, pp. 431–435, doi:10.1145/775832.775946.

[10] Alexander Kaiser, Daniel Kroening & Thomas Wahl (2010): *Dynamic Cutoff Detection in Parameterized Concurrent Programs*. In: *Computer-Aided Verification (CAV)*, doi:10.1007/978-3-642-14295-6_55.

[11] T. Kam, T. Villa, R. Brayton & A. Sangiovanni-Vincentelli (1998): *Multi-valued decision diagrams: theory and applications*. *Multiple-Valued Logic* 4(1-2), pp. 9–62.

[12] K. L. McMillan (1993): *Symbolic Model Checking*. Kluwer Academic Publishers, Norwell, MA, USA.

[13] A. S. Miner (2001): *Efficient solution of GSPNs using Canonical Matrix Diagrams*. In: *Proceedings of the 9th International Workshop on Petri Nets and Performance Models*, IEEE Comp. Soc. Press, pp. 101–110, doi:10.1109/PNPM.2001.953360.

[14] A. S. Miner (2004): *Saturation for a General Class of Models*. In: *QEST '04: Proceedings of the The Quantitative Evaluation of Systems, First International Conference*, IEEE Computer Society, Washington, DC, USA, pp. 282–291, doi:10.1109/QEST.2004.38.

[15] G. L. Peterson (1981): *Myths About the Mutual Exclusion Problem. Inf. Process. Lett.* 12(3), pp. 115–116, doi:10.1016/0020-0190(81)90106-X.

[16] A. Pnueli (1981): *A temporal logic of concurrent programs*. *Theoretical Computer Science* 13, pp. 45–60.

[17] A. Pnueli, S. Ruah & L. Zuck (2001): *Automatic Deductive Verification with Invisible Invariants*. Springer, pp. 82–97.

[18] J.-P. Queille & J. Sifakis (1982): *Specification and verification of concurrent systems in CESAR*. In: *DesProceedings of the 5th Colloquium on International Symposium on Programming*, Springer-Verlag, London, UK, pp. 337–351, doi:10.1007/3-540-11494-7_22.

[19] R. K. Ranjan, A. Aziz, R. K. Brayton, B. Plessier & C. Pixley (1995): *Efficient BDD algorithms for FSM synthesis and verification*. In: *In IEEE/ACM Proceedings International Workshop on Logic Synthesis, Lake Tahoe (NV*.

[20] F. Somenzi (2009): *CUDD: CU Decision Diagram Package, Release 2.4.2*. University of Colorado at Boulder, http://vlsi.colorado.edu/ fabio/CUDD/.

[21] T. Wahl, N. Blanc & A. Emerson (2008): *Sviss: Symbolic Verification of Symmetric Systems*. In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*.

[22] B. Yang, R. E. Bryant, D. R. O'Hallaron, A. Biere, O. Coudert, G. Janssen, R. K. Ranjan & F. Somenzi (1998): *A Performance Study of BDD-Based Model Checking*. In: *Proceedings of the Second International Conference on Formal Methods in Computer-Aided Design*, FMCAD '98, Springer-Verlag, London, UK, pp. 255–289.